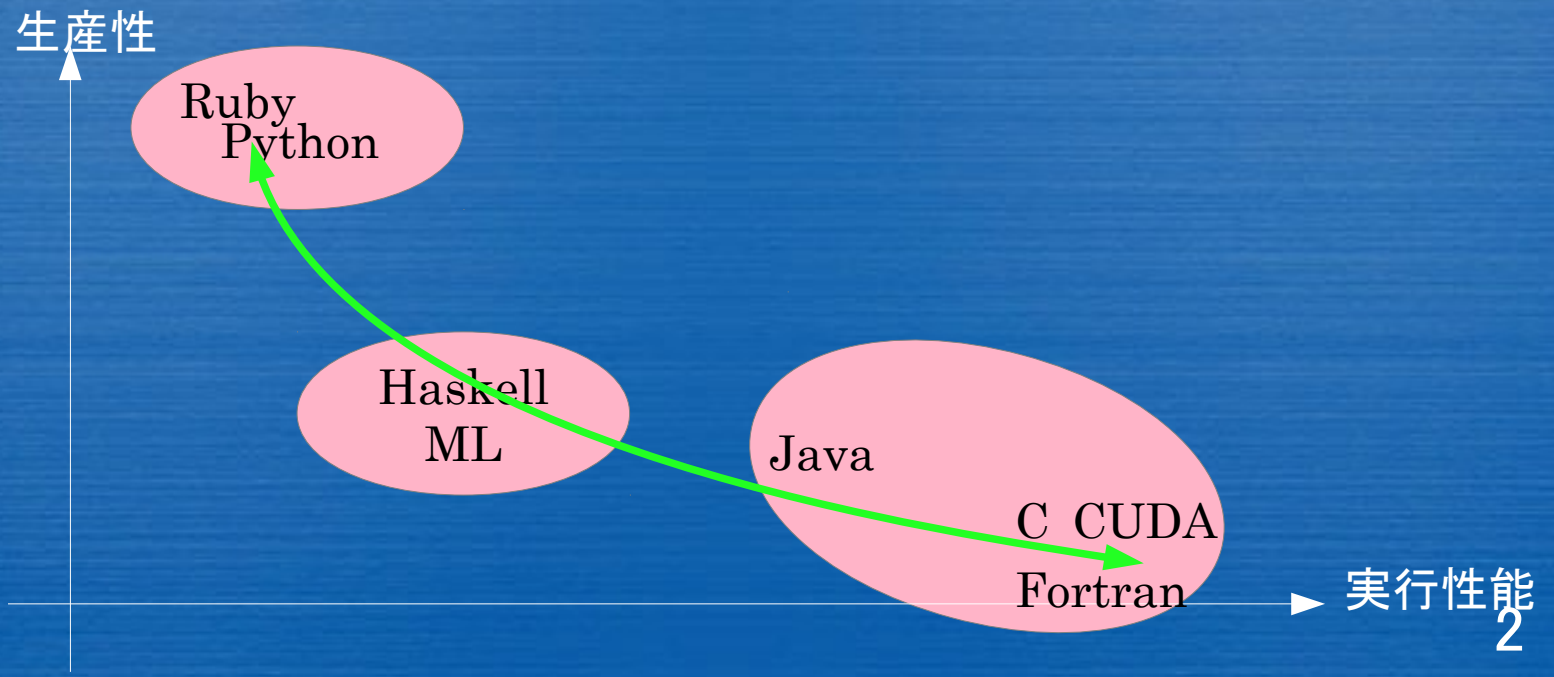
多種言語処理系性能の評価に適したベンチマークプログラム

東京大学
野瀬貴史，泊久信，平木敬

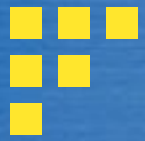
本研究の目的

- プログラムを作るときにどんな言語で作るか？
- 「実行性能」と「生産性」のバランス
- これまでは実行性能と生産性の両方について客観的指標を得ることが出来なかった
- 本研究では性能に関する評価を行う



概要

- 言語同士の，根拠を持たせた
  性能比較をベンチマークにより行う
  - 手動による移植ではなく自動変換を行う
  - 変換するベンチマークに実績のあるものを用いる
- このためにトランスレータを作成した
- 26種の言語処理系のデータを取り比較できるようになった



既存研究

- Computer Languages Benchmark Game
  - 人力での移植
  - 規模の小さなプログラムの集まり
    - 一番大きいものでもJavaで180行
  - そもそも真面目な研究ではないと明言
  - 実績のあるベンチマークとの比較が困難

SPECは非現実的

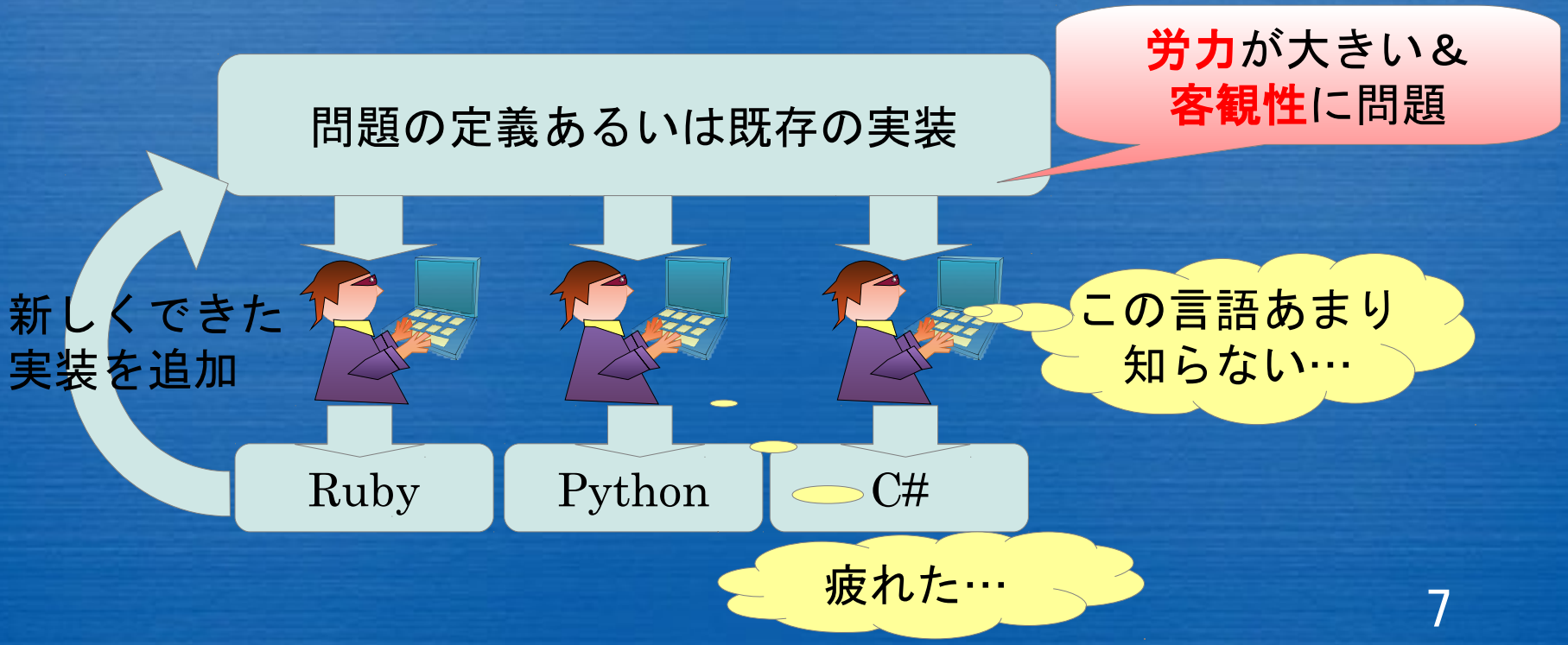
- 実績のあるSPECを使うのが望ましいが，実行時間が長いため，遅い言語向けに移植した際の実行時間が非現実的な長さになる
- NAS Parallel Benchmarks (NPB)とDhrystoneはSPECとの良い相関が示されている[泊2011]
  - 実行時間が短い
  - NPBのプログラムの大きさはそれぞれ500行～5000行程度で平均3100行
- NPBとDhrystoneに基づく比較が適切

NPBとDhrystone

- NAS Parallel Benchmarks:
  - IS, CG, LU, FT: 数値計算カーネル
  - BT, MG, SP: 数値流体力学
- Dhrystone: 整数演算性能の測定．文字列操作が多い
- SPEC2006:
  - CFP: 流体力学，MD，レイトレ
  - CINT: 言語処理，圧縮アルゴリズム

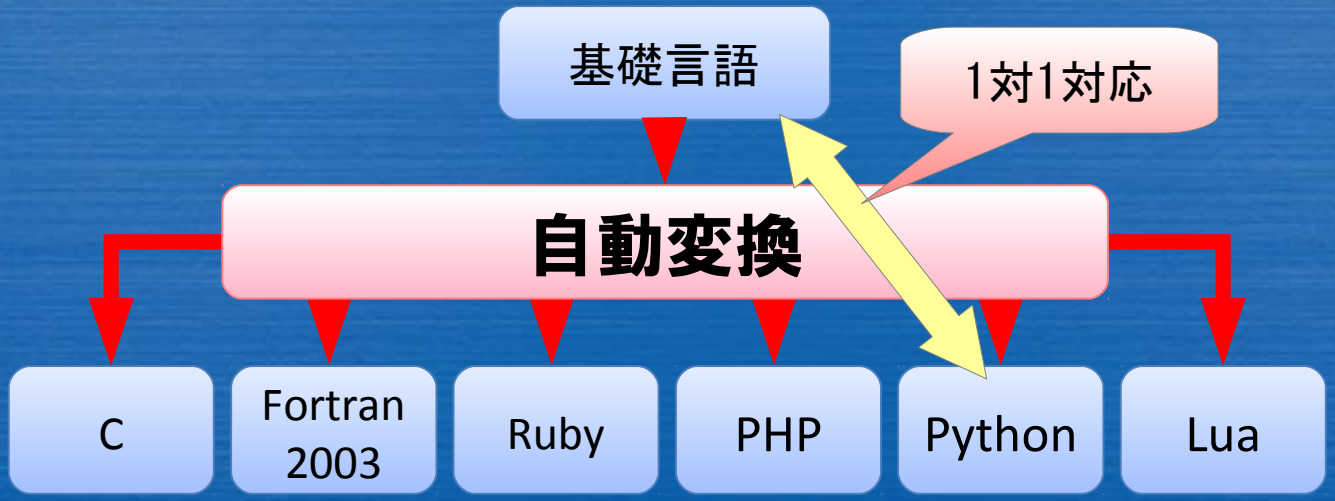
手動移植の問題点

- 労力がかかりすぎ，規模を拡大できない
- 実装者の言語への習熟度により
  チューニングにばらつきが出る



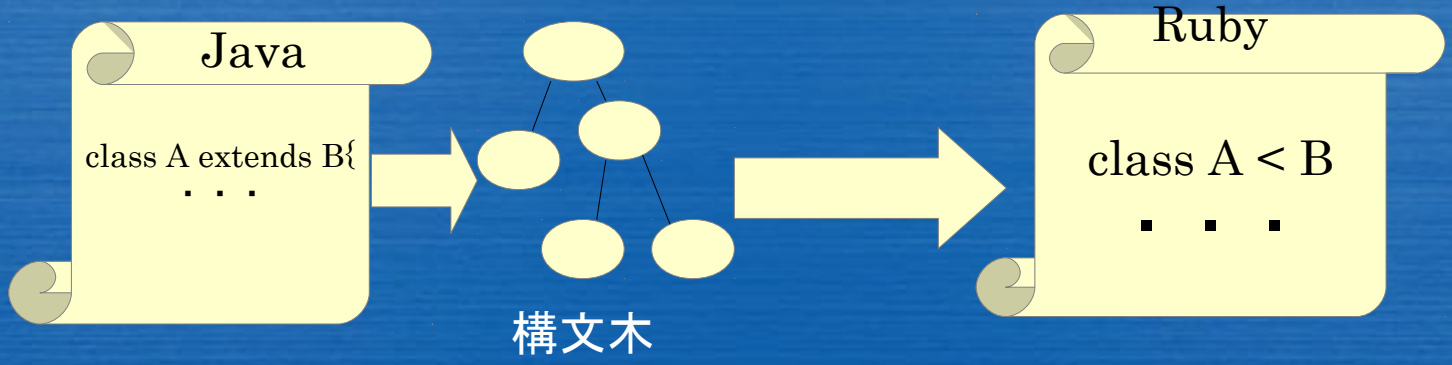
提案手法: 自動変換器の使用

- 自動トランスレータを使って作業を自動化する
  - 意味論の1対1対応が取れる
    - 実装者の得意・不得意による不確実性が排除
  - 労力の大幅削減



トランスレータの実装

- 比較の起点となる言語と変換ルールを合わせるためNPBとDhrystoneのJava版を使用
- Javaの構文木から直接ターゲット言語のソースを生成



実装の方針

- 制御構造・配列・クラスなどの概念は今回対象にしている言語はどれも持つので概ね1対1変換可能
- トリッキーな記述への変換はなるべく避ける
- 変換ルールの例外をなるべく抑える

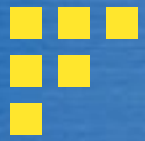
変換の例: もとのソース

```
public class CG extends CGBase {
  (中略)
  public double getMFLOPS(double total_time,int niter){
    double mflops = 0.0;
    if( total_time != 0.0 ){
      mflops = (float)( 2*na )*
     ( 3.+(float)( nonzer*(nonzer+1) )
     + 25.*(5.+(float)( nonzer*(nonzer+1) ))+ 3. ) ;
      mflops *= niter / (total_time*1000000.0);
    }
    return mflops;
  }
}
```

変換の例: Fortran 2003 (1/2)

```
module m_CG
use m_CGBase

implicit none
  type, extends(t_CGBase) :: t_CG
  contains
    procedure, pass(this) :: init => CG_init
    (中略)
    procedure :: getMFLOPS => CG_getMFLOPS
  end type t_CG
contains
(続く)
```

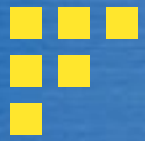


変換の例: Fortran 2003 (2/2)

```
double precision function CG_getMFLOPS(this, total_time,
niter)
  class(t_CG), intent(inout) :: this
  double precision, intent(in) :: total_time
  integer, intent(in) :: niter
  double precision :: mflops = 0.0
  if (total_time /= 0.0d0) then
      mflops = dble((2 * this%na)) * (3.0d0 + dble((this
%nonzer * (this%nonzer + 1))) + 25.0d0 * (5.0d0 +
dble((this%nonzer * (this%nonzer + 1))))  + 3.0d0)
      mflops = mflops * niter / (total_time * 1000000.0d0)
  end if
  CG_getMFLOPS = mflops
  return
end function CG_getMFLOPS
end module m_CG
```

変換の例: Ruby

```
class CG < CGBase
  (中略)
  def getMFLOPS(total_time, niter)
    mflops = 0.0;
    if total_time != 0.0 then
        mflops = (2 * @na) * (3.0 + (@nonzer * (@nonzer +
1)) + 25.0 * (5.0 + (@nonzer * (@nonzer + 1))) + 3.0)
        mflops *= niter / (total_time * 1000000.0)
    end
    return mflops
  end
end
```



変換の例: Python

```
class CG(CGBase):
  (中略)
  def getMFLOPS(self, total_time, niter):
      mflops = 0.0;
      if total_time != 0.0:
          mflops = float((2 * self.na)) * (3. +
float((self.nonzer * (self.nonzer + 1))) + 25. * (5. +
float((self.nonzer * (self.nonzer + 1)))) + 3.)
          mflops *= niter / (total_time * 1000000.0)
      return mflops
```

行数がどれくらい変化するか

CG.javaの変換前と変換後の比較

Java	C99	F03	Ruby	Python	PHP
587	574	681	557	415	508

※1. すべてコメントを除いた行数
　　コメントの復元は今回行わない
※2. C99は
　　CG.cの行数：492
　　CG.hの行数：82

変換ルール:For文

- 変換できないFor文はWhile文へ

```
//変換前の Java ソース
for(i=isize-1,j=0;i>=0;i--,j++){
   処理内容 ();
}
```

```
#変換後の Ruby ソース
i = isize - 1
j = 0
while i >= 0 do
   処理内容 ()
   i -= 1
   j += 1
end
```

変換ルール:オーバーフロー

- PHPでは整数同士の演算でオーバーフローすると結果がFloatになる

```
<?php
$million = 1000000;
$large_number =  50000 * $million;
var_dump($large_number);  // float(50000000000)
?>
```

変換ルール:オーバーフロー

- Ruby，Pythonでは多倍長整数になる

```
irb(main):001:0> a = 1000000 * 50000
=> 50000000000
irb(main):002:0> a.class
=> Bignum
```

Ruby

```
 >>> a = 1000000 * 500
 >>> type(a)
 <type 'int'>
 >>> a = 1000000 * 5000
 >>> type(a)
 <type 'long'>
```

Python

変換ルール:オーバーフロー

- PHPでは整数同士の演算でオーバーフローすると結果がFloatになる
- Ruby，Pythonでは多倍長整数になる
- 整数演算のオーバーフローへの対処はトランスレータ側では困難
  - ベンチマーク側をオーバーフローが起きないよう式変形して対処

変換ルール:配列の確保

- 配列のサポート度合いは言語によってまちまち
  - 多次元配列を確保する関数が無かったり
  - 配列ではなくリストだったり(一括での確保ができない)
- 言語によっては多次元配列の場合
  独自の配列確保ルーチンを呼び出す

```
#大きさ 3x2x5 の配列を確保
a = MultiDimArray(3,2,5)
```

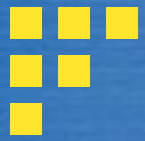
変換ルール:Cでの”継承”

- クラスAを継承したクラスBがある場合
- struct Bではstruct Aのメンバと同じものを並べたあとB固有のメンバを並べる
- 使うときは無理やり型キャスト
- アドホックではあるが変換元が型チェックを通ったプログラムなので動作に問題はない

性能評価

変換した言語および測定した処理系一覧

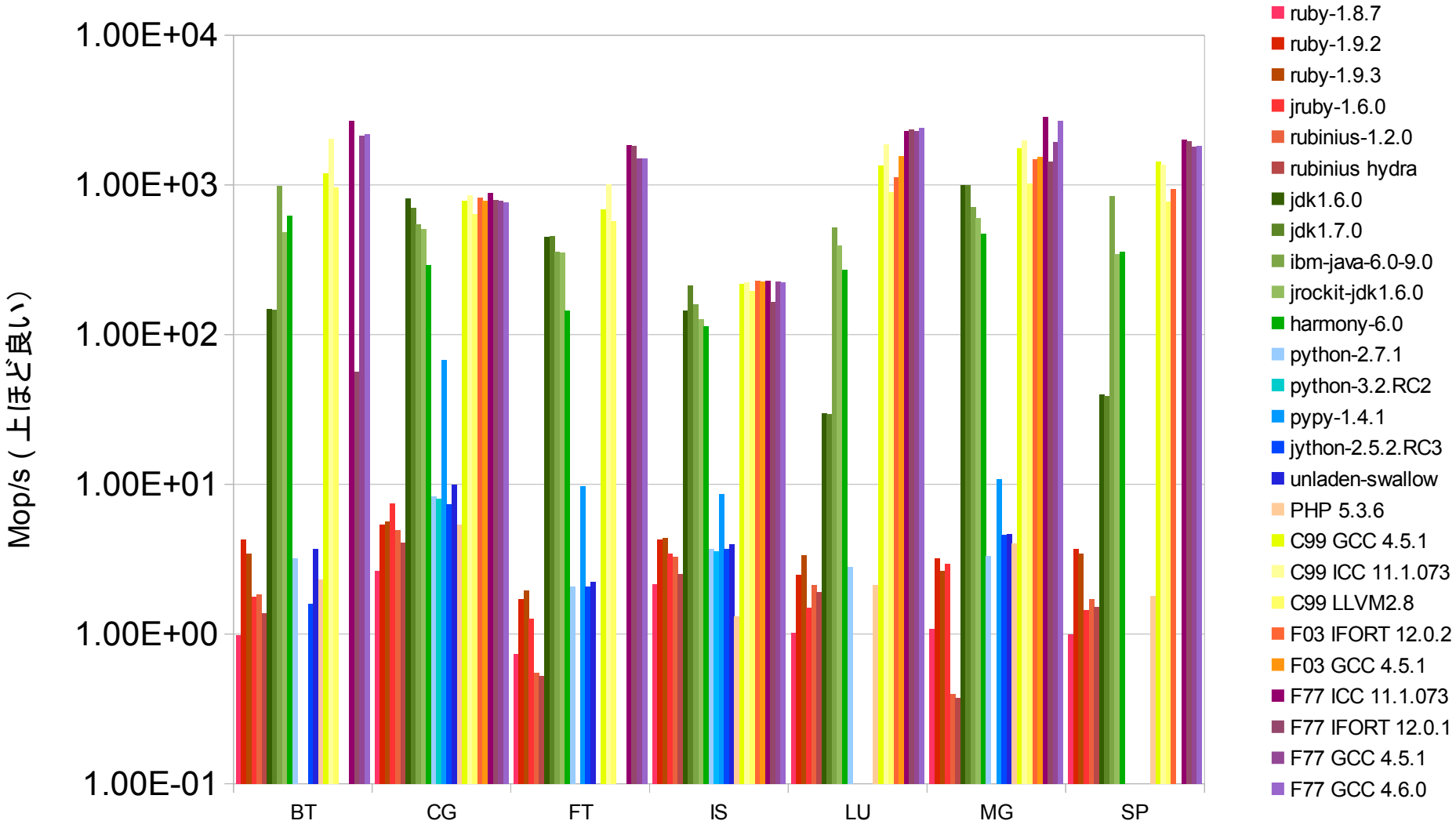
Java	Sun JDK 1.6.0_23
	Sun JDK 1.7.0-ea-b127
	IBM Java 6.0-9.0
	Oracle JRockit JDK 1.6.0_20-R28.1.0-4.0.1
	Apache Harmony-6.0-jdk-991881
Ruby	Ruby 1.8.7-p330
	Ruby 1.9.2-p136
	Ruby 1.9.3-110206
	JRuby 1.6.0.RC1
	Rubinius 1.2.0
	Rubinius 1.3.0dev hydra
Python	Python2.7.1
	pypy 1.4.1
	Jython 2.5.2.RC3
	unladen-swallow (--without-llvm)
PHP	PHP 5.3.6
C99	GCC 4.5.1 (-O3 -msse4.2)
	ICC 11.1.073 (-fast -xSSE4.2)
	LLVM 2.8(-O3 -msse4.2)
Fortran 2003	IFORT 12.0.2 (-fase xSSE4.2)
	GCC 4.5.1 (-O3 -msse4.2)



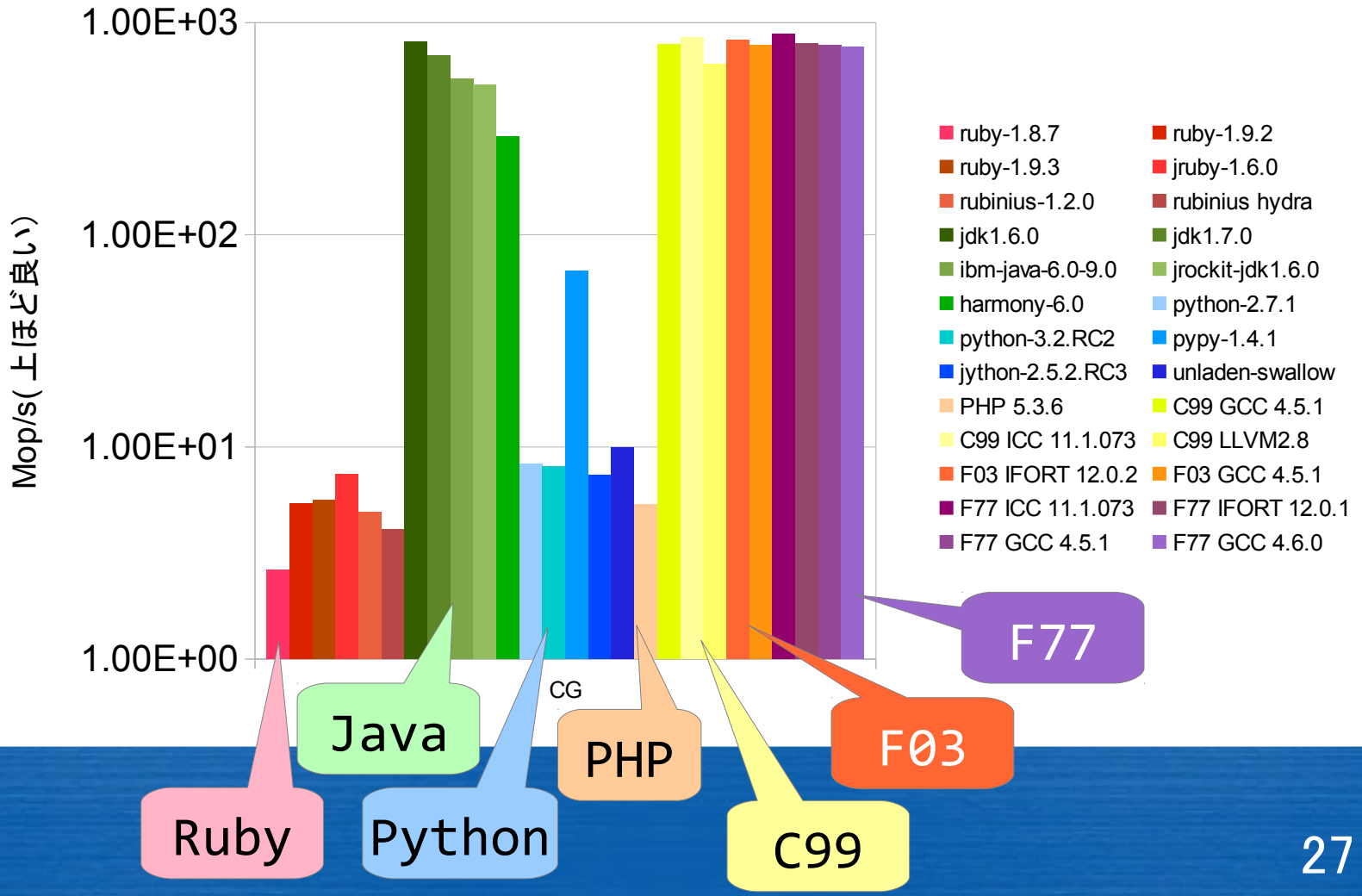
環境

- Dell PowerEdge R410
  - Xeon E5530 2.4GHz
  - CentOS 5.5
  - メモリ12GB
- NPBの問題サイズはA
- Dhrystoneの反復回数は50,000,000
- 言語処理系のコンパイルはGCC 4.5.1
  - 最適化オプション：-O3 -msse4.2

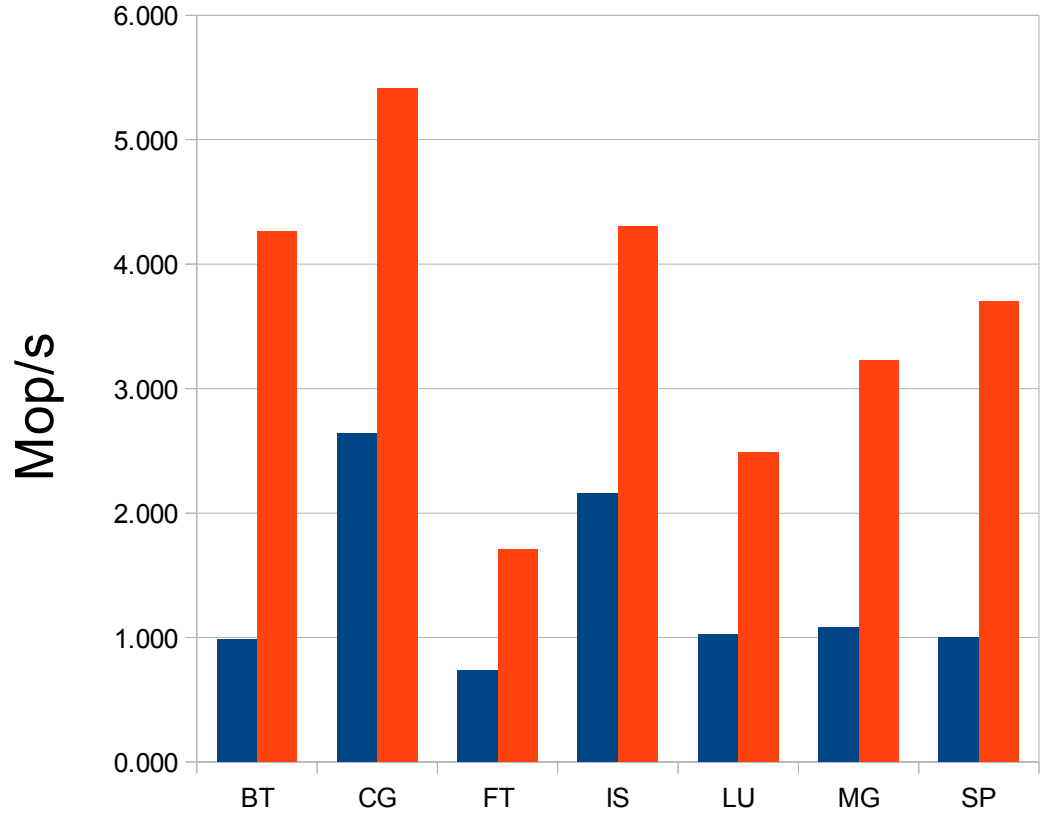
NPBすべて（対数目盛）


Mop/s（上ほど良い）
1.00E+04
1.00E+03
1.00E+02
1.00E+01
1.00E+00
1.00E-01
BT
CG
FT
IS
LU
MG
SP
ruby-1.8.7
ruby-1.9.2
ruby-1.9.3
jruby-1.6.0
rubinius-1.2.0
rubinius hydra
jdk1.6.0
jdk1.7.0
ibm-java-6.0-9.0
jrockit-jdk1.6.0
harmony-6.0
python-2.7.1
python-3.2.RC2
pypy-1.4.1
jython-2.5.2.RC3
unladen-swallow
PHP 5.3.6
C99 GCC 4.5.1
C99 ICC 11.1.073
C99 LLVM2.8
F03 IFORT 12.0.2
F03 GCC 4.5.1
F77 ICC 11.1.073
F77 IFORT 12.0.1
F77 GCC 4.5.1
F77 GCC 4.6.0

CG（対数目盛）

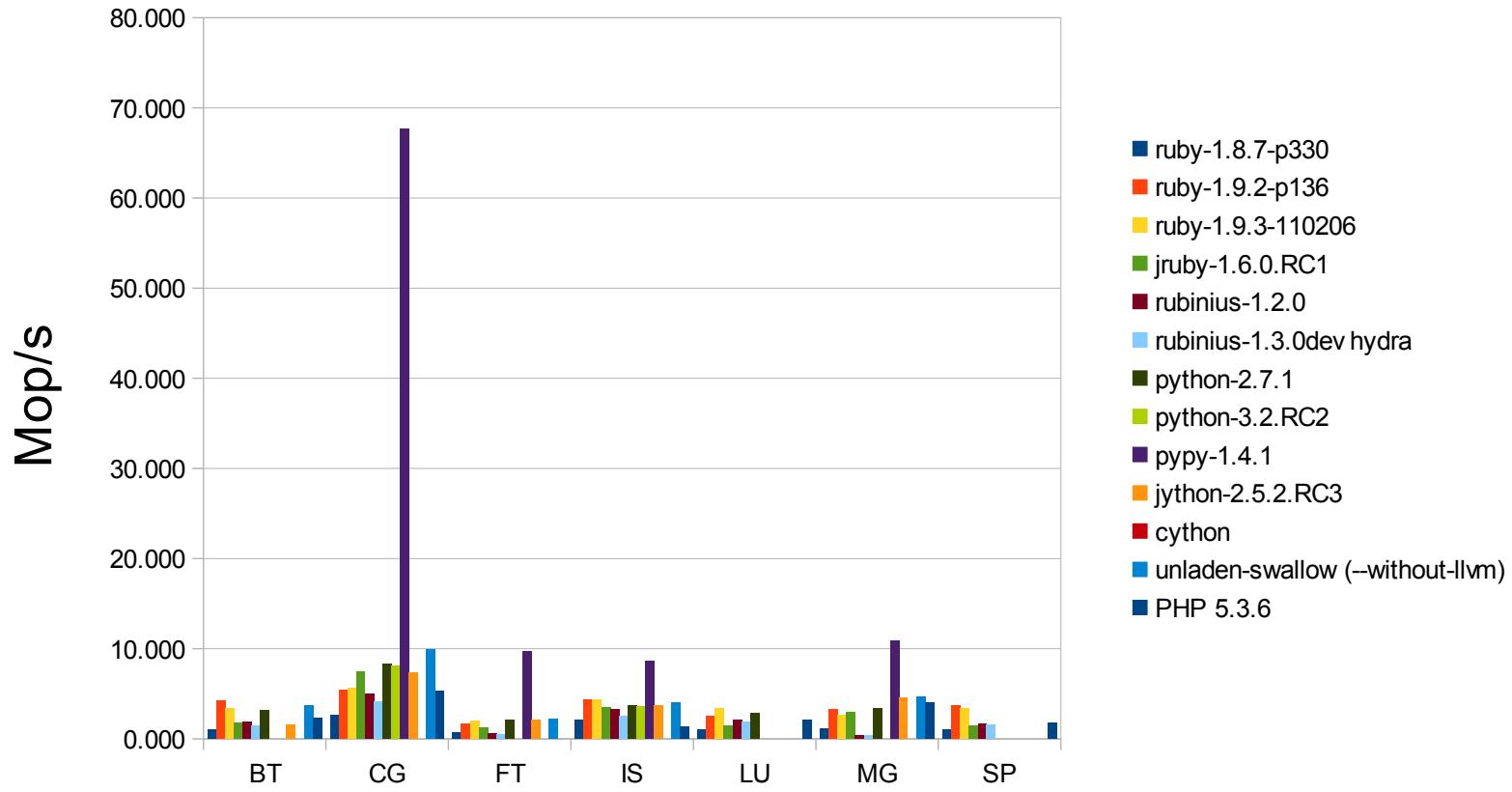

1.00E+03
1.00E+02
1.00E+01
1.00E+00
Mop/s(上ほど良い)
CG
ruby-1.8.7
ruby-1.9.2
ruby-1.9.3
jruby-1.6.0
rubinius-1.2.0
rubinius hydra
jdk1.6.0
jdk1.7.0
ibm-java-6.0-9.0
jrockit-jdk1.6.0
harmony-6.0
python-2.7.1
python-3.2.RC2
pypy-1.4.1
jython-2.5.2.RC3
unladen-swallow
PHP 5.3.6
C99 GCC 4.5.1
C99 ICC 11.1.073
C99 LLVM2.8
F03 IFORT 12.0.2
F03 GCC 4.5.1
F77 ICC 11.1.073
F77 IFORT 12.0.1
F77 GCC 4.5.1
F77 GCC 4.6.0
F77
F03
C99
PHP
Python
Java
Ruby

Ruby



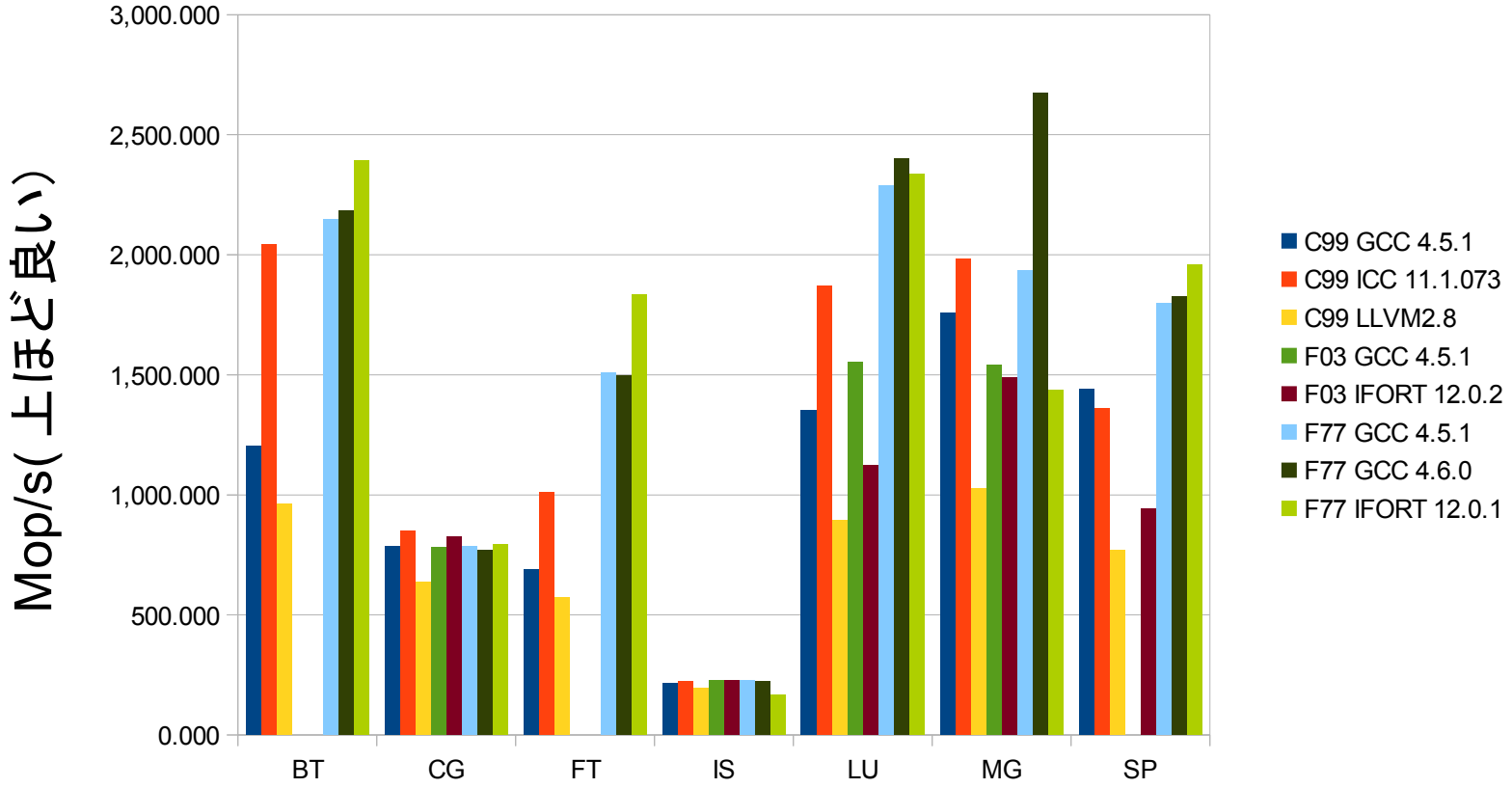
安定して
2倍以上の差

Ruby/Python/PHP

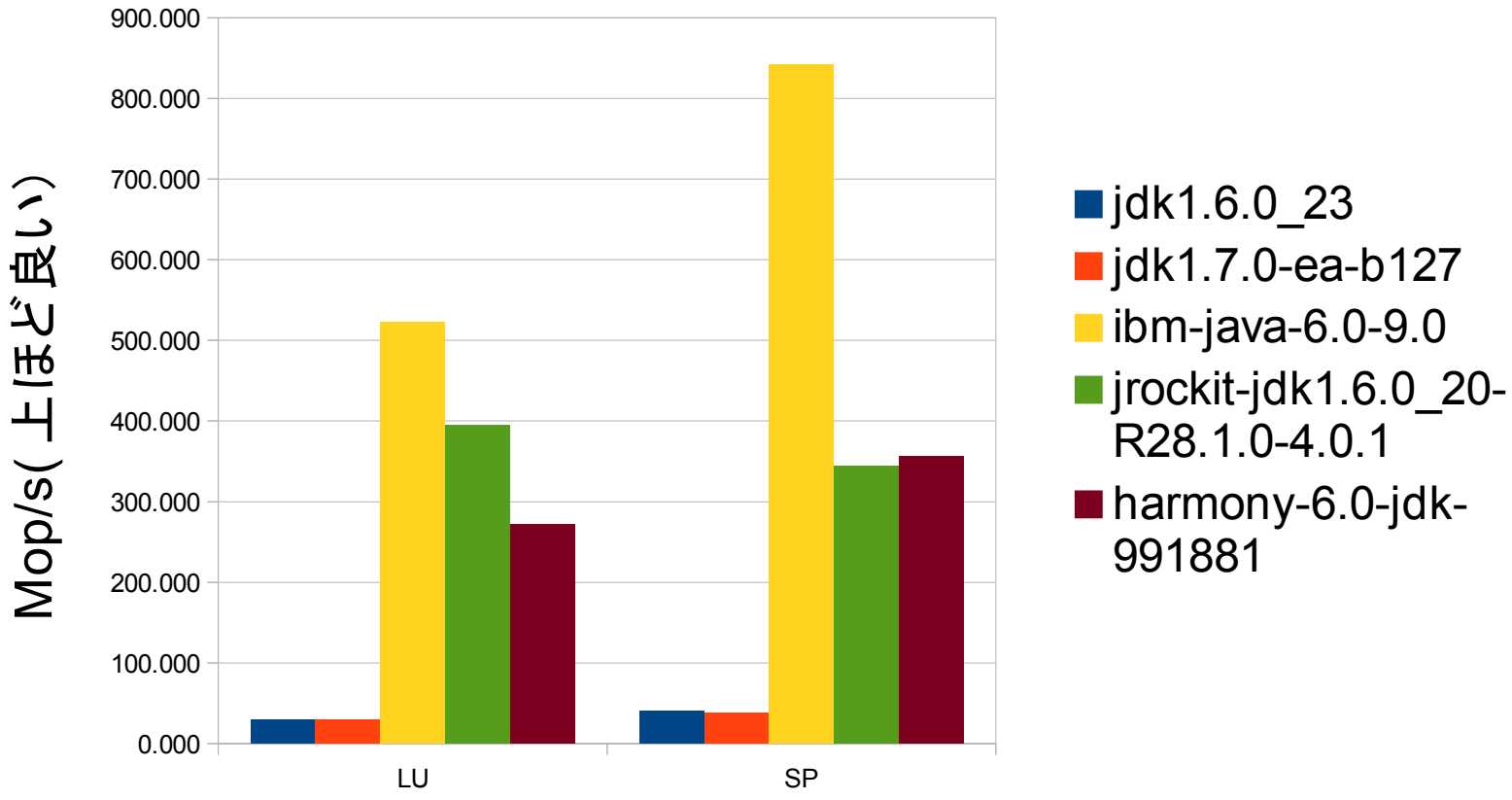


CGのPypy/Python2.7比は8.16倍

ネイティブコンパイルされる言語

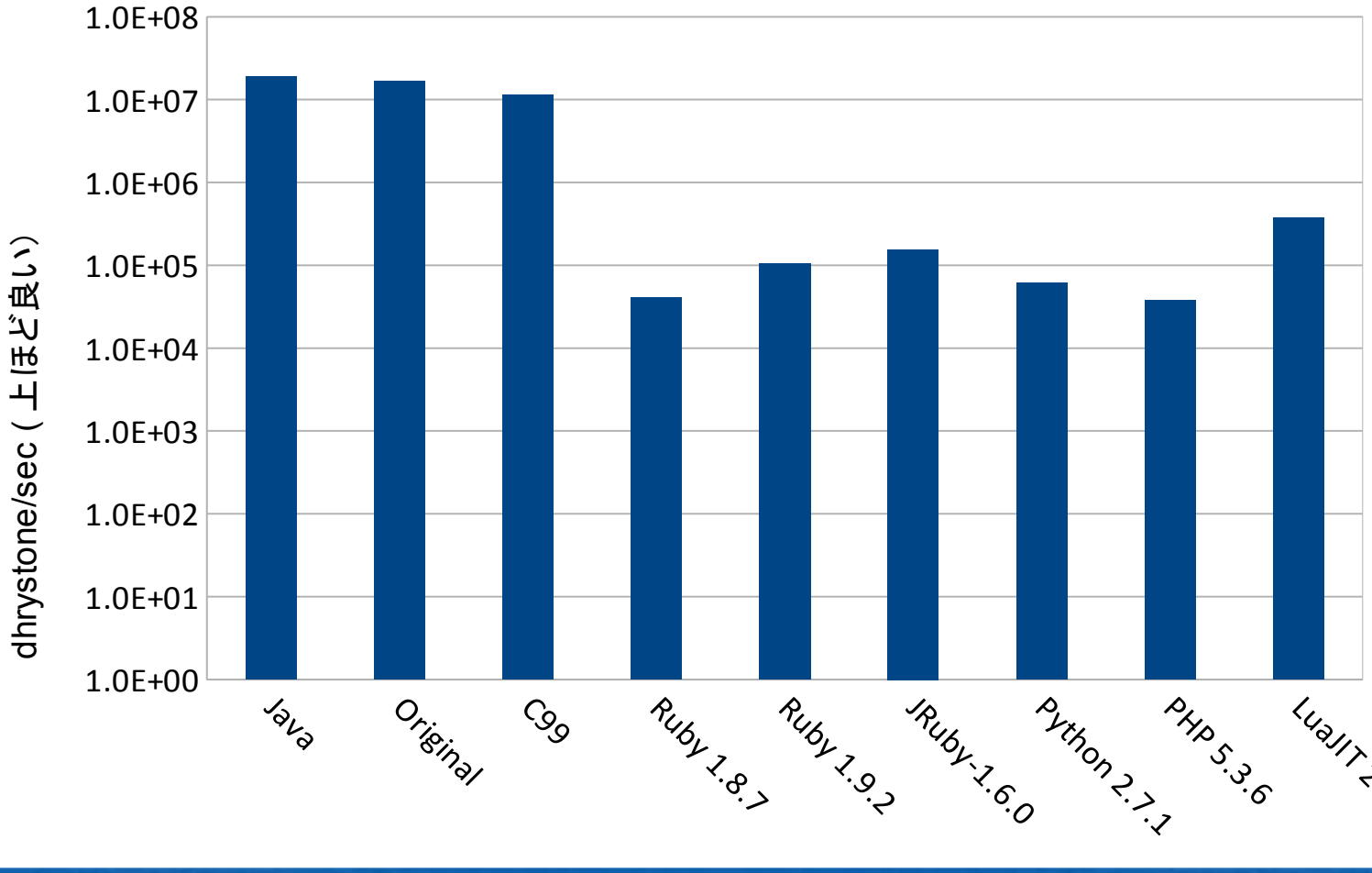


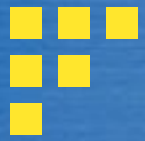
JavaVMの実装ごとの性能差



JDK1.6とIBM JavaでLU, SPそれぞれ17.5倍, 21.0倍の差がある

Dhrystone


dhrystone/sec（上ほど良い）
1.0E+08
1.0E+07
1.0E+06
1.0E+05
1.0E+04
1.0E+03
1.0E+02
1.0E+01
1.0E+00
Java
Original
C99
Ruby 1.8.7
Ruby 1.9.2
JRuby-1.6.0
Python 2.7.1
PHP 5.3.6
LuaJIT 2.0.0 beta

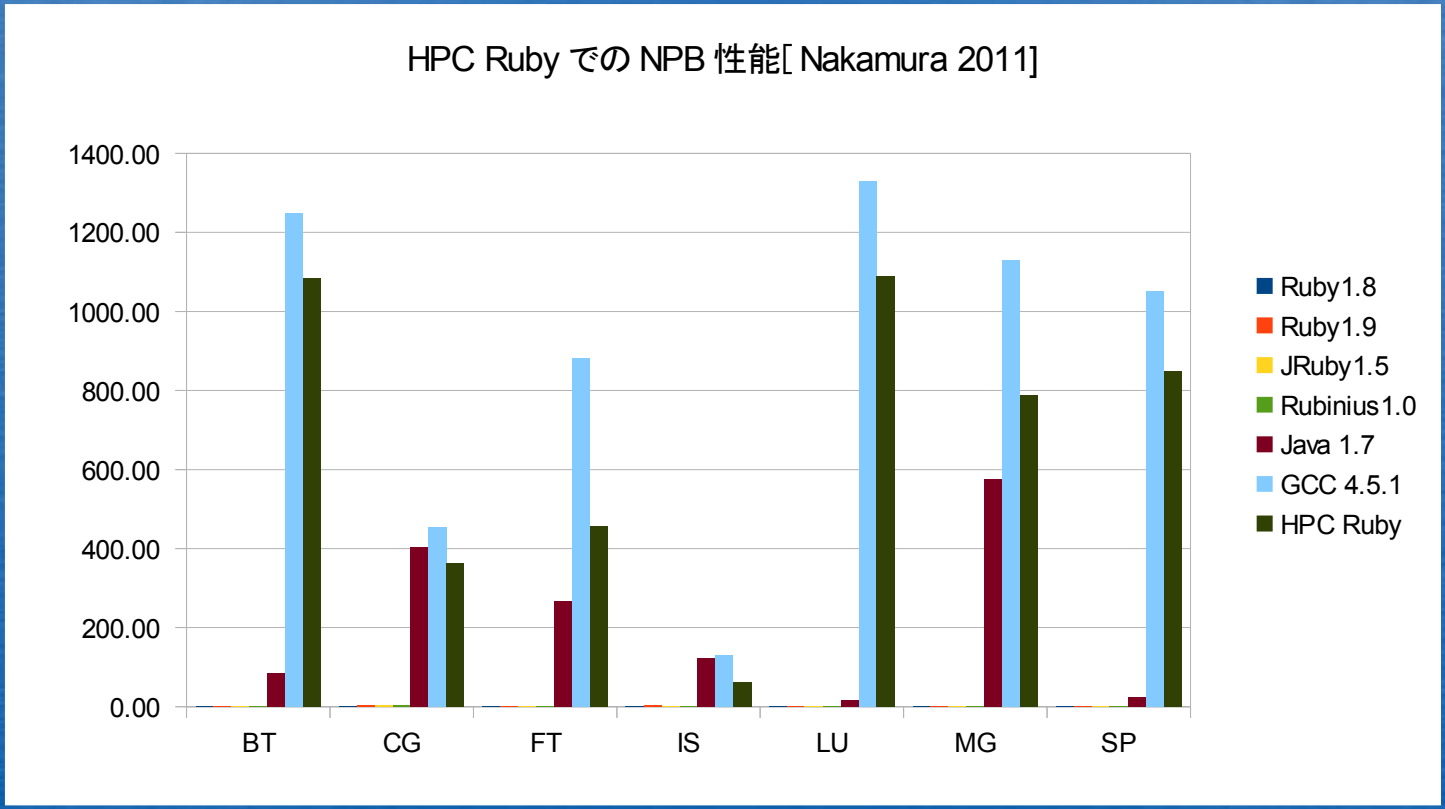


考察

- SunのJVMの以外の性能はおおむねIBM, JRockit, Harmonyの順であるが，Sunは安定しない
  - この種の数値計算には不向き
- ネイティブコンパイルする言語ではISの性能に差がつかない
  - I/Oバウンド
  - CGも似た傾向
- Fortran 2003はCG/ISを除くとF77の半分程度の性能に下がる
- PHPは遅いイメージを持たれているが致命的なほどではない

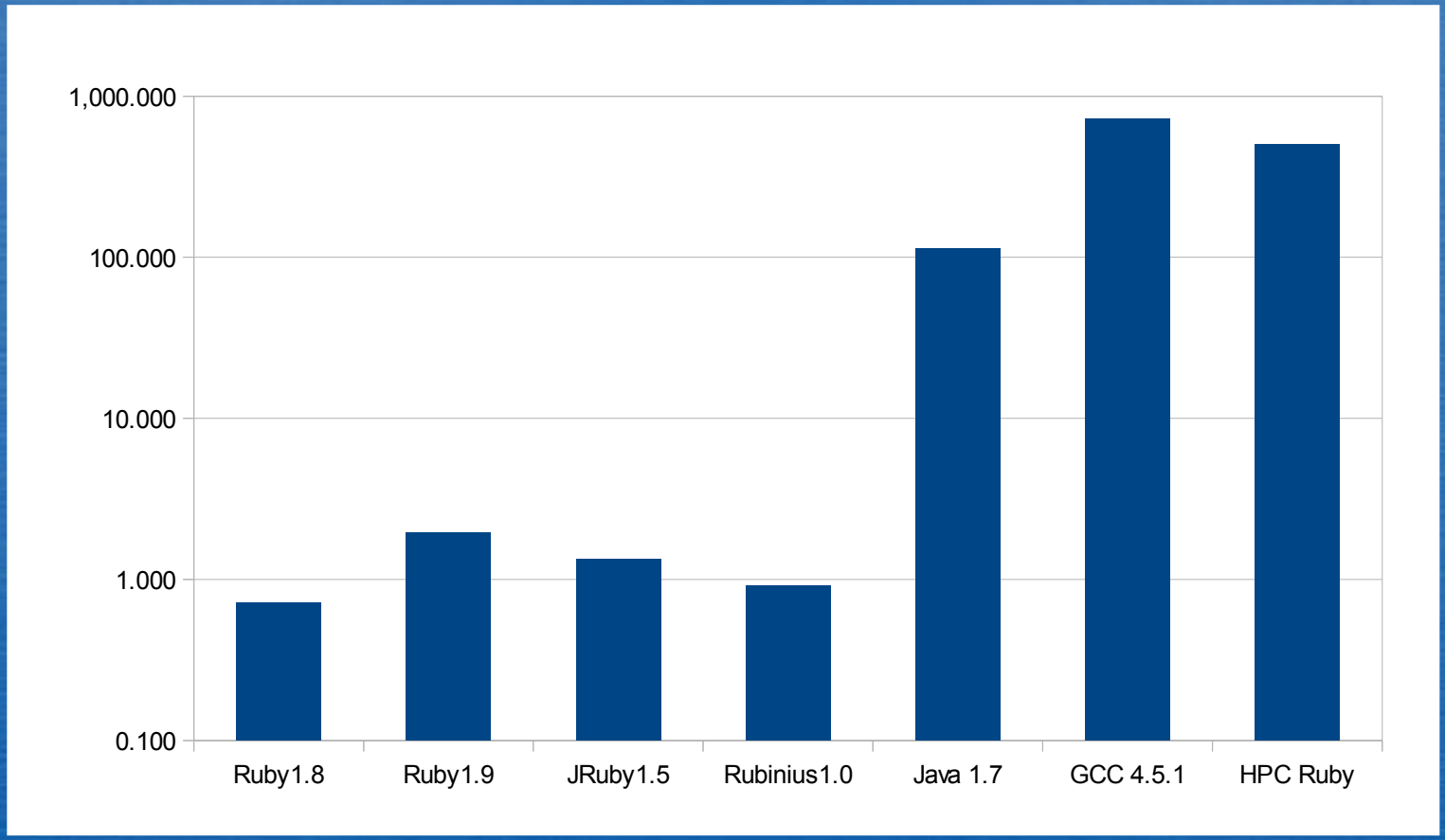
HPC Rubyの性能評価

- 平木研が開発しているHPC Rubyの性能測定に既に使われている



HPC Rubyの性能評価

- スコアの幾何平均（対数目盛）



まとめ

- 言語の生産性と性能のトレードオフの勘案をするために性能測定による比較が必要
- 公平性の確保のため
  - トランスレータを使用
  - SPECと相関のあるNPBとDhrystoneを変換
- 26種の言語処理系のデータを取り比較できるようになった
- すでに実際に言語処理系の評価に使われている

今後の課題

- 測る言語・言語処理系の拡大
- 演算性能以外の測定項目の拡大
- 基礎ベンチマークがJavaに最適化されているのでさらに言語中立な記述にする
- SPECjvm2008を取り入れる
- より高級な機能で実装されたベンチマークを新たに作成する